NotSoOld After fixing fac_irrupt and chain_pick and chain_find (and some other... 9ce89b5 22 hours ago

1 contributor

1488 lines (991 sloc) 61.7 KB

```
  ____                 _______  _______
 / __ \___  ___ ___   / ___/ _ \/ __/ __/
/ /_/ / _ \/ -_) _ \ / (_ / ___/\ \_\ \
\____/ .__/\__/_//_/\___/_/  /___/___/
    /_/           by NotSoOld, 2017 (c)

        route|process|gather stats
```

OpenGPSS Manual (release 1.0)

Navigation

General

Program in OpenGPSS language looks like following:

```
*definition area*
*exit condition*
{{
executive area
}}
*another definition area if needed*
{{
another executive area
}}
...
...
```

Definiton area contains definitions of variables, facilities, queues and marks that are used to simulate a system. Every definition line is like:

- For variables:

```
type name = initial_value;
```

- For structures:

```
type name {initial parameters};
```

Every separate line in OpenGPSS finishes with ';'. If the line doesn't have ';' at the end, next line is recognized as the continuation of current line. Comments are C-like: `// This is single line comment`

```
/* And this is multiline
comment */
```

When simulation is going, exit condition is tested to know when simulation should be stopped. Exit condition should be defined only once:

```
exitwhen(expression with boolean result);
```

When this expression turns to true, simulation will stop.

Double brackets `{{` and `}}` separate executive area from definition area. Executive area is an area where xacts are added and removed, move and process. Executive area contains executive blocks:

```
optional_mark_name:executive_block_name(block params);
```

assignments to variables or xact parameters (and increments/decrements):

```
optional_mark_name:var_name = new_value;
```

```
optional_mark_name:var_name += expression;
```

```
optional_mark_name:var_name++;
```

and single braces for *if/else_if/else/while/loop_times* blocks.

Xact parameters can be accessed like this:

```
xact.p1, xact.str5, xact.my_parameter, xact.priority
```

(*xact.priority* is a special parameter which is used by interpreter to drive some logic, so every xact has it by default.)

Every line in executive area (except curly braces) can start with name of the mark followed by mark separator. In other words, presence of mark separator in the line means that xact can be transported to this line. Curly braces **cannot** be addressed, it will lead to errors.

If xact reaches some executive line, it tries to execute it (except single curly braces and *inject* block - it executes automatically) using its own parameters if needed.

Nearly every parameter - queue/facility name, *wait/travel/if* parameters - can be not just words, but complex expressions. They will be parsed to a string/number before block execution. (Except parameters in definitions and *inject* block, because these are parsed before any execution of blocks starts. But initial values for variables and array/matrix size can be expressions.)

Model while simulating has two very important lists: *future events chain*, FEC, and *current events chain*, CEC. Time in model is measured in beats (so, it's discrete). Every beat FEC is watched if there are xacts that need to move in the current beat. If they do, they are moved from FEC to CEC. Then, CEC is sorted according to xacts' priority, and every xact in CEC is moving through executive blocks until it will be a) rejected from the model b) blocked c) moved to a user chain d) executing *wait* block.

In case a) xact will be deleted from CEC.

In case b), which can be caused by trying to enter busy facility or *wait_until* block with failed condition, xact will remain in CEC up to the next beat, in which it tries to move again.

In case c) xact will be removed from CEC and added to one of user chains.

In case d) xact will be moved to FEC with exit time (time when it needs to move further) set by *wait* block.

New xacts can be added to the model through *inject* and *copy* blocks. Inject block presents an *injector* - it adds one xact to FEC with exit time set according to injector's parameters, and when this xact leaves FEC, it sends a signal to injector that it's time to inject one more xact into FEC. *Copy* block creates copies of xacts which are added into CEC.

There is a special situation called *CEC review*. When interpreter receives a signal "review CEC", it interrupts movement of current xact and starts to go through CEC from its beginning. Blocks like *fac_leave*, *review_cec* and changing xact's *priority* can trigger CEC review (because these actions can affect simulation process. For example, if some xact leaves facility, it becomes available for xacts which wait at *fac_enter* block but cannot proceed. And when CEC will be reviewed, xacts will be able to move to unlocked facility. Changing of priority may affect the order of xacts' processing.)

Definition types

For all types:

Name of the variable (as string) can be accessed through dot: *variable.name*.

There is also indirect addressing (tilde sign, "~"), which allows to get the value of variable *which* name is in other variable, for example:

```
str var1 = 'buffered';
int buffered = 5;

~var1++; <== will increment variable "buffered".
```

Also:

Current xact (which executes block, assignment, etc.) has some accessible parameters along with its own parameters. They are accessed through dot operator:

```
xact.index
xact.group
```

Simple variables:

- int

  Just a variable which can hold an integer value and be accessed by its name. Range is the same as range of integer in Python.

  Notes:

  - If a float value is assigned to int variable, it is implicitly converted to int.

  - There are three read-only default integer variables which are used by interpreter and can be accessed by user:

  injected - count of xacts injected by injectors of the system;

rejected - count of xacts rejected from the system through *reject* blocks;

curticks - how long simulation is going.

- float

  Just a variable which can hold a floating point value and be accessed by its name. Range is the same as range of float in Python.

- str

  Just a variable which can hold a string value and be accessed by its name. Strings can be accessed, assigned and concatenated only.

- bool

  Just a variable which can hold a boolean value (true/false) and be accessed by its name. Boolean variables can be only assigned or accessed and can be used in logic expressions.

Structure types:

- fac (facility)

  This is a device which can be occupied by xacts. Usual practice is to use facility to simulate something that can be occupied by somebody for some amount of time and therefore make other transacts wait until it'll be freed. So, when facility is free (has spare places), xacts can occupy it and move further, but facility is fully busy, xacts will wait until it'll have free places.

  Initial parameters (can be set in curly braces):

  - isQueued = bool *(default == true)* - if true, this facility will be automatically queued as if there are *queue_enter* and *queue_leave* blocks around *fac_enter* block. Queue will be named with this facility's name.

  - places = int *(default == 1)* - how many xacts can occupy this facility until it becomes busy.

  Accessible parameters (through dot operator):

  - curplaces - how many free places are currently available

  - maxplaces - how many places facility has at all

  - enters_f - how many xacts entered the facility at this time

  - isAvail - current availability status of facility

- queue

  This is a device which is generally used for gathering statistics about the flow of transacts near facilities. But queueing can be used not only around *fac_enter* blocks. Statistics include number of entered xacts, current xacts in the queue, etc. It is important to mention that queues doesn't really sort or queue xacts, it's only gather statistics.

  Initial parameters: none.

  Accessible parameters (through dot operator):

  - curxacts - how many xacts are currently queued

  - enters_q - how many xacts entered the queue at this time

- mark

  This is a definition of a transporting mark which can be used in the executive area of a program. When mark is at the left of ':' in the line, it defines a line where xacts should be transporting when this mark is mentioned. When mark is present as argument of transport operator (or somewhere else where xacts can be moved around the model), it declares where xact should go, which line it should follow after execution.

  Initial parameters: none.

  Accessible parameters: none.

  Additional notes:

- You will always see messages about undefined or unused marks.

- chain

  User chains are used to store transacts here when you need to control their flow through the model. User chains enable user to buffer xacts (and to simulate buffering devices), to release xacts one by one at some point in the model, etc.

  Initial parameters: none.

  Accesible parameters (through dot operator):

  - length - how many xacts are currently in this user chain

  - xacts (only available inside find/find_minmax functions!) - list of current xacts in the chain

- hist<variable_name> (histogram)

  Histograms are one of the ways to gather statistics. Histogram can collect value of a one parameter during simulating. Parameter name is specified in <> brackets after *hist* keyword. Value of this parameter is added to the histogram by calling *hist_add* block. After simulating, histogram will be printed in both text and pseudo-graphical representations.

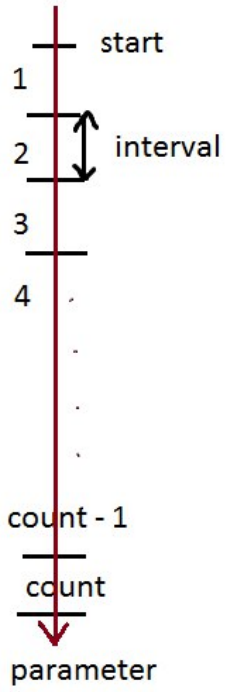
  Initial parameters (all of them **must** be set in curly braces):

  - start - it is first bounding value of histogram

  - interval - it is a constant interval between histogram marks

  - count - total number of intervals (excluding interval from -infinity to start and from last mark to +infinity).

  Here is graphical representation of these parameters:

  When parameter value is about to be added to histogram, according interval will be chosen. Each interval contains not value of the parameter, but a number of parameter value additions of this interval.

  Accessible parameters (can be accessed thorugh dot operator):

  - enters_h - how many samples have been added to histogram (weighted)

  - average - average value of observing parameter

- graph<x_var, y_var> (2D graph)

  Graps are another way of gathering statistics about changes of variables' values. This type allows to gather information in the form of two-dimensional graph (i.e. one value is X, and another is Y on the plot, and they are stored as pairs). According to math laws, every X value can be associated only with one Y value; so, when sampled (X, Y) pair already exists, it will be saved as (X, (oldY + Y) / 2).

Since plot building is not possible using pseudographics, output of a graph in simulation results will look like table of X-Y pairs, so you can copy them and build plot in Excel, etc.

Initial parameters: none.

Accessible parameters: none.

Arrays and matrices:

Variable of **every** type (simple or structural, except *mark* and *function*!) can be defined not as a single variable, but as array:

```
int arr[10] = 0;
queue CPUs[4];
fac CPUs[4] { isQueued = false, places = 1 };
```

or as matrix:

```
float my_matrix[[5, 6]];
bool adjacent[[7, 7]] = false;
```

Initial values specified after "=" or in "{}" will be applied for every array/matrix element. Arrays' length and matrices' width and height are constant (defined only once). Three- and more dimensional arrays are not supported.

Inside, arrays' and matrices' elements are defined as separate objects *(with names like "array_name&&index" and "matrix_name&&(index, index)")* but at the end of simulation they will be printed either with proper names ("array[index]", "matrix[[index, index]]") (for structure types) or definitely as arrays and matrices (for simple types).

Conditional functions:

These functions which can be defined by user are a feature to replace GPSS's FUNCTION block. There is a word "conditional" in the name, because this function consists of pairs "*condition, result*" and, from first to last, condition of every pair will be evaluated. First condition which occurs to be true will command the interpreter to return result which was in the pair with this condition.

So, here is a prototype:

```
function function_name(arguments, if any) {
                                          condition1, result1 |
                                          condition2, result2 |
                                          ...
                                          conditionN, resultN
                                         };
```

Every condition is an expression with boolean (or something which can be converted to boolean) result; every result can be an expression. Condition and result are separated by comma, pairs are separated by "|". If no condition succeeds, function will return 0 (if you need to override default behaviour, it's recommended to leave final pair as "*1, value_to_return_if_every_condition_fails*", so it will be always returned if every other condition failed).

Function also can have some arguments which values will be put to condition expressions and result expressions when function is called. Please **do not** name parameters as your system variables, etc., because then they will be overwritten by function parameters. (But names like "p1" are allowed, even if function uses expressions like "xact.p1".)

Simpliest example of "abs" implementation:

```
function my_abs(value) {
                        value >= 0, value |
                        value < 0, -value     // or "1, -value", because it will always succeed
                       };
```

Attachable Python function modules

OpenGPSS has a special feature for Python programmers or simply for people who like to write and use complex computational functions - these functions can be described in Python language and attached as a module with use of *attach* keyword in definition area:

```
attach calculus;
...
// Somewhere in execution area:
xact.str1 = calculus.derivative("x^2");
// etc.
```

It means that in the folder where *OpenGPSS Interpreter.py* is there is "calculus.py" file and inside "calculus.py" file there is Python function "def derivative(function_string):" or something like that. There can be as many functions as you want, just be aware their names are unique. You also can attach as many modules as you want. Important: attachable modules **should** be in the same folder where interpreter is (i.e. in root folder of this project). Relative and absolute paths are not supported, only one-word names of modules.

This feature is especially useful when these functions cannot be described using built-in OpenGPSS features.

About name ambiguity

OpenGPSS is a case-sensitive language. Names can consist of upper and lower register letters, digits (but these names cannot **start** with digit) and underscores. Exceptions are strings ("like this") and comments - they both can contain any ASCII symbols. Escape characters in strings are supported.

In some cases identical names are allowed:

- names of structures with different types can be the same (queue CPU, mark CPU, fac CPU)

- names of arrays and names of variables can be the same (int myvar, int myvar[10], int myvar[[5, 4]])

But:

- **do not** name variables with identical words, they'll be messed up (little explanation: structures are either used as blocks' arguments (in this case, their names are parsed as strings, so, there's no big difference what type this structure was of, if name is correct for block) or their accessible parameters are accessed by dot operator (and different parameters have different names for each structure type, as you can mention). And variables can be accessed just like "my_variable_name = ...", and you cannot say, what variable is accessed here if you have multiple of them with different types)

- **never** name any variables/functions/structures as keywords (including "xact", "chxact", "curticks", "injected", "rejected", etc.)

- Always check that names of your own functions/variables/structures do not match with names of built-in functions and functions from attached modules!

- Also check that names of modules do not match with keywords or any variables/structures in the system.

Remember, these rules are always a subject to change.

Interpreter configuration file

There is special file called *opengpss_config.cfg* in the root folder (if there is no config file, start and close interpreter, file will appear with default values in it). This file allows to control aspects of logging - printing debug information while simulating - and some other cool features. Value after "=" shows if feature is enabled or disabled; so, you can enable and disable them.

List of available features (with defaults in parens):

- enable_nice_vt100_codes (True) - enables using of VT100 formatting tags (they allow to print bold/italic/coloured text). Disable it if you aren't working in Linux terminal (Windows doesn't know what these tags mean).
- results_to_file (False) - if true, results of simulation will be printed to file named "name_of_model_results_date_of_simulation.txt"; otherwise all output goes into console. If file with results already exists, new file will be created (without removing old file).
- log_to_file (False) - if true, all output connected with debugging during simulation will be printed to file "name_of_model_log_simulation_date.txt"; otherwise all debug output goes to console. If log file already exists, new file will be created (without removing old file).
- print_program_in_tokens (True) - if true, before simulation your program will be printed as interpreter sees it, i.e. in tokenized form (tokens are separate numbers, words, operators, blocks, etc.). Sometimes useful for debugging misspellings.

- log_tick_start (True) - if true, at the beginning of every beat the message with the current tick index will be printed.
- log_CEC_and_FEC (True) - if true, in the beginning of every beat CEC and FEC contents will be printed. CEC is also printed every time CEC review is triggered.
- log_xact_trace (True) - if true, every xact will print name and line of block which it currently tries to enter, so you can see trace of each xact.
- log_xact_blocking (True) - if true, every time xact is blocked corresponding message will be printed.
- log_facility_entering (True) - if true, every time xact occupies some facility corresponding message is printed.
- log_FEC_entering (True) - if true, every time xact executes wait() block, a message will be printed (with time when xact will leave FEC).
- log_assignments (False) - if true, every assignment parsing result will be printed. Use it when you're not sure if assignment is interpreted correctly.
- log_dot_operator (False) - if true, you'll see what left-side and right-side values of each dot operation are.
- enable_antihalt (True) - "antihalt" is a special "watchdog": if there is nothing happening in the system for a long time (no xact movement, no xacts in CEC and FEC), a dialog will appear which asks if the simulation should be finished now as it won't go any further. System can halt not only because of user's fault, system might be organized in a such way. So, in some cases you might want to disable antihalt.
- antihalt_threshold (1000) - threshold value for antihalt watchdog, measured in ticks. If CEC and FEC are empty for specified amount of ticks, a dialog will appear.
- tick_by_tick_simulation (False) - if true, in the begginning of every time beat the simulation will wait user's input (so you can read logs of previous beat simulation); if false, simualtion will go from beginning to end.
- block_by_block_simulation (False) - similar to parameter above, but simulation will stop before each block execution (so it will be very slow).

Executive blocks

inject - add xacts into your system

- Prototype:

```
inject(
         string xact_group_name,
         int time,
         int timedelta,
         int initdelay,
         int inject_limit
        )
        {
         parameter1 = int/float/boolean/string,
         ...
         (here go all your parameter definitions)
         ...
         priority = int/float (default == 0)
        };
```

- Usage:

  This block will add an xact of group *xact_group_name* every *time* beats until it reaches its *inject_limit*. Xacts will start moving from the line where *inject* block stands. Time between injections can be modified: first xact can be delayed by *initdelay* beats, and *time* parameter can be randomized to *time*±*timedelta* with even distribution.

  {parameters} are optional (if you don't use them, just leave no braces or empty braces). In OpenGPSS, you must define each parameter you want to use further in the simulation. There are no limitations for names (except cases written in "About name ambiguity" section; and name cannot start with digit). Each parameter will have its type defined by its initial value, and this type **should** be preserved. Types are int, float, boolean, and string. *priority* is a special number parameter to define priority to xacts. Priority can be used for controlling the order of a processing, etc. Xact parameters can be accessed using keyword *xact* followed by dot and parameter name.

- Example:

```
inject("main", 10, 4, 0, 250) {p1 = 0, one_more_param = 'go', priority = 10};
```

- Additional hacks:

  - If *inject_limit* equals zero, there is no limit for this *inject* block.

  - *timedelta* and *timedelay* also can be zeros.

  - If *inject_limit* is positive, but parameters *time* and *timedelta* are zeros, *inject_limit* xacts will be added simultaneously. (It is not a special work mode of *inject* block but a feature which works because of block's code structure; so, you might be careful - if *inject_limit* is also zero in this situation, the system will hang.)

  - Parameters with matching names will be overwritten (the latest parameter will be saved only).

  - Be attentive while naming parameters, or you'll get "current xact does not have a parameter named "..." " errors very quickly.

queue_enter - enter unordered queue to gather statistics

- Prototype:

```
queue_enter(
             word queue_name
            );
```

- Usage:

  This block will queue executing xact in the queue *queue_name*.

- Example:

```
queue_enter(CPU);
```

- Additional hacks:

  - Queue name can be an expression with string result (for example, name of one of the queues).

  - Xact cannot enter queue in which he already is, so be careful.

queue_leave - leave previously entered unordered queue

- Prototype:

```
queue_leave(
             word queue_name
            );
```

- Usage:

  This block will remove current xact from queue *queue_name*. If xact will try to leave queue which it didn't enter, an error will be raised.

- Example:

```
queue_leave(CPU);
```

- Additional hacks:

  - Queue name can be an expression with string result (for example, name of one of the queues).

  - Xact cannot leave queue in which he is not present, error will be raised.

fac_enter - occupy facility by taking one of its free places

- Prototype:

```
fac_enter(
          word fac_name,
          int xact_volume (default == 1)
         );
```

- Usage:

  This block is used to simulate a facility which can be occupied by some number of xacts. If facility *fac_name* has free places, xact will move further and will be present in facility's busyness list. If facility is fully busy, xact will stop at this block and will try to enter this facility again every beat until it proceeds (this condition has name - "blocked"). Xact can occupy more than one facility 'seat', it can be set by *xact_volume* argument.

  fac_enter is usually queued by *queue_enter* and *queue_leave* blocks to gather iformation about how long xacts wait to enter busy facility and how much of them are waiting.

- Example:

```
fac_enter(CPU);
```

- Additional hacks:

  - Facility name can be an expression with string result (for example, name of one of the facilities).

  - Facilities can be automatically queued (so, you won't need to write queueing blocks manually) as if it is *queue_enter* block right above *fac_enter* block and *queue_leave* block right under *fac_enter* block.

  - If xact tries to enter facility which it already occupies, an error raises.

fac_leave - free a place in previously occupied facility

- Prototype:

```
fac_leave(
          word fac_name
         );
```

- Usage:

  This block will free a place in facility *fac_name*, triggering CEC review to give an ability to blocked xacts to occupy freed facility. If the facility was not previously occupied by leaving xact, an error will be raised.

- Example:

```
fac_leave(CPU);
```

- Additional hacks:

  - Facility name can be an expression with string result (for example, name of one of the facilities) (and you an also do this trick with queue names, chain names, etc. I won't mention it anymore below).

  - This block automatically triggers interpreter to review CEC.

  - Xact cannot leave facility which he is not occupying.

fac_irrupt - force into occupied facility

- Prototype:

```
fac_irrupt(
           word fac_name,
           int xact_volume (default == 1),
           bool eject (default == false),
           word mark (default == next after this block),
```

```
 word elapsedto (default == none)
);
```

- Usage:

  Sometimes you need not to just occupy free facility, but to come into it, stop processing of occupying xacts and/or occupy facility by yourself. For example, a person busy with something can be interrupted from signals or people coming outside.

  So, *fac_name* can be interrupted by xact with *xact_volume* (how many places xact needs in facility for itself; it will try to move xacts only from this amount of places). If *eject* is false, currently processing xacts will be taken from model chains (CEC, FEC, user chains) and moved to facility interruption chain. When *eject* == false, there are no more arguments. No ejection means that xacts from interruption chain will continue processing after interrupting xact goes away with the help of *fac_goaway* block. These xacts will process as much time as they have to process when they were interrupted.

  When *eject* == true, it means that interrupted xacts will be ejected from this facility and won't return to processing automatically. If *mark* is present, they'll be sent there. If *elapsedto* is present (it is a name of some variable/parameter), elapsed processing time wil be written there.

- Examples:

```
fac_irrupt(CPU, 1, True, to_elapsed, xact.p3);
fac_irrupt(fac, 1, True, '', xact.p1);
fac_irrupt(fac, 3, False);
```

- Additional hacks:

  - If xact is too big (in terms of volume) to irrupt (i.e. enter) the facility, it will be skipped (*move()* block is called).

  - If facility is fully free, *fac_enter()* will be called instead.

  - If facility is not available, it cannot be irrupted (*move()* block will be called).

  - Xact cannot irrupt facility which it has already occupied or irrupted.

  - **Always** pair this block with *fac_goaway()* block if you send xacts to interruption chain, or they'll never leave it.

  - In fact, there is no huge difference between irrupting and entering facility; the only thing is if there are no free places and you try to enter facility, you'll be blocked, but if you'll try to irrupt it, you'll succeed. Xact inside facility does **not** have a status (like "I am legally here" or "I interrupted this facility").

fac_goaway - go away from previously interrupted facility

- Prototype:

```
fac_goaway(
           word fac_name
          );
```

- Usage:

  After you interrupt facility, you need to go away from it (especially when you push some xacts into interrupt chain - otherwise they'll be there forever!). When interrupting xact passes this block, it deletes itself from facility and moves xacts to freed places from interrupt chain.

- Example:

```
fac_goaway(CPU);
```

- Additional hacks:

  - If xact is not occupying or hasn't irrupted this facility, it is OK to enter this block, *move()* function will be called (no CEC review in this case!).

  - This block sends signal to interpreter to review CEC.

fac_avail - make facility available for xacts

- Prototype:

```
fac_avail(
          word fac_name
         );
```

- Usage:

  This block has an ability to turn availability status of facility back to "available".

- Example:

```
fac_avail(CPU);
```

- Advanced info:

  - Every facility is available by default.

  - When facility is available, it means in can be entered (if is has free places) and interrupted.

  - This block sends a signal to review CEC.

fac_unavail - make facility closed for everyone

- Prototype:

```
fac_unavail(
            word fac_name
           );
```

- Usage:

  Sometimes you need to simulate unavailability of a facility, so that's why this block is implemented.

- Example:

```
fac_unavail(CPU);
```

- Advanced info:

  - When facility is unavailable, it means in **cannot** be entered or interrupted. When facility status turns to "unavailable", it's like facility will be fully busy for incoming xacts (i.e. they will be blocked); xacts processing in this facility will leave it when processing finishes (as usual). If you want to "close" facility immediately, irrupt it first and then make it unavailable.

  - *fac_leave* and *fac_goaway* blocks will still work as usual if facility is unavailable.

reject - delete xact entirely from system

- Prototype:

```
reject(
       int reject_counter_inc
      );
```

- Usage:

  This block is used to delete xacts from system. It means that xact will not move through model anymore and will be erased from CEC. *reject_counter_inc* defines how much should be added to reject counter default variable (*rejected*).

- Example:

```
reject(1);
```

- Additional hacks:

  - If you are so inattentive that you send xacts to block *reject* which are in some facilities/queues, interpreter will automatically erase these xacts from corresponding facilities/queues.

  - This block automatically triggers interpreter to review CEC (because of previous hack - after deleting xact from facility this facility can be occupied by other xacts).

wait - move xact to FEC for some amount of time

- Prototype:

```
wait(
     int time_to_wait,
     int time_delta (default == 0)
    );
```

- Usage:

  This block is used to simulate processing of a xact. It moves xact to FEC (preventing it from moving through model immediately) setting its exit time to *time_to_wait*±*time_delta* (with even distribution).

- Example:

```
wait(8, 3);
```

transport family blocks ("->>", "->|", "->?") - transport xact or fork the path of xact

- Prototypes:

```
->> word markname;
->| word markname_if_true,
    float probability,
    word markname_if_false (default == block next to this block)
->? word markname_if_true,
    expression condition,
    word markname_if_false (default == block next to this block)
```

- Usage:

  These blocks are used to transport xacts from one point in model to another. Transportation can be unconditional (->>), depend on probability (->|) or depend on condition (->?). You can also determine where xacts should go if probability/condition check fails (by default they will just go further through the model).

- Examples:

```
->> CPU_mark;
->| Mark1, 0.4, jmp;
->? mark, xact.pr > 10;
```

- Additional hacks:

  - If you don't like "->" notation, you can use following block names: *transport()*, *transport_prob()*, *transport_if()*.

if/else_if/else - make xact follow different paths according to some condition

- Prototypes:

```
if(
   expression condition
  )
{
        blocks which will be executed in condition == true
}
else_if(
        expression another_condition
```

```
       )
{
        blocks which will be executed if another_condition == true
}
else_if...
...
else
{
        what to do if every other chained conditional blocks above failed
}
```

- Usage:

  These blocks are used to execute different parts of a program. Choice is made according to conditions in parens - group of blocks in curly braces with a first true condition will be chosen to execute. **Curly braces cannot be omitted!**

  There can be many *else_if()* blocks or no of them; also, *else* block can be omitted. Conditional blocks are considered chained and are tested as a whole thing if they are written as in the example, one right after another.

- Examples:

```
if(xact.f3 > 0)
{
        fac_enter(CPU1);
}
else_if(myChain != 3)
{
        fac_enter(CPU2);
}
else
{
        reject(0);
}

if(boolVar)
{
        chain_purge(ch1, toTerm);
}
else
{
        chain_enter(ch1);
}

etc.
```

- Additional hacks:

  - These conditional blocks can be nested. Use them as you'll use them in C or any other similar language.

wait_until - block xact movement until condition becomes true

- Prototype:

```
wait_until(
           expression condition
         );
```

- Usage:

  Sometimes you don't need xacts to move through special part of program until something happened. So, you can prevent xacts from doing that by using this block. If xact enters this block and condition fails, it will remain staying on this block until condition turns to true. Condition is checked every beat.

- Example:

```
wait_until(buffer.length > 0);
```

chain_enter - move xact to one of user chains

- Prototype:

```
chain_enter(
            word chainname
           );
```

- Usage:

  User chains are very powerful when you need a mechanism to buffer xacts in one place of the model and, when needed, take them out to another place. This block simply chains xact into *chainname* chain. New xact will be moved from CEC to the end of user chain.

- Example:

```
chain_enter(buffer);
```

- Additional hacks:

  - Current xact block will be this block (even if xact is in the chain).

chain_leave - take xacts from user chain

- Prototype:

```
chain_leave(
            word chainname,
            int count,
            word where_to_move (default is next block after that)
           );
```

- Usage:

  You can move xacts from user chains back to your model with the help of *chain_leave* block. *Count* xacts (or less, if there aren't enough xacts in the chain) will be taken from chain *chainname* and moved to the *where_to_move* mark in the model (or if there is no mark, they'll be moved one block further after *chain_leave*).

- Example:

```
chain_leave(buffer, 2, tofacility);
```

chain_purge - take all xacts from the user chain

- Prototype:

```
chain_purge(
            word chainname,
            word where_to_move (default is next block after that)
           );
```

- Usage:

  It is equal to `chain_leave(chain, chain.length, block)` . The chain will be empty after calling this block.

- Example:

```
chain_purge(buffer, killmark);
```

chain_pick - take xacts which satisfy a condition

- Prototype:

```
chain_pick(
           word chainname,
```

```
           condition,
           int count,
           word where_to_move (default - next block after this)
          );
```

- Usage:

  While *chain_leave* gives an ability to unconditionally remove xacts from user chains, *chain_pick* allows to take only xacts which meet specified condition. Condition may look like "chxact.parameter ==/<=/... some value or expression" (it's just an example; left side also may vary). New keyword *chxact* represents xact from user chain which is currently observed (when checking condition, every xacts from user chain will be observed, from first to last). *count* represents how many times should condition be checked; if chain has less suitable xacts than *count*, it is OK.

  This block is important because it gives a possibility to remove xacts from chains according to their parameters' values, it may be used in some situations (and it cannot be made in other way, I think).

- Examples:

```
// Take 3 xacts which "ptime" parameter is more than 10
chain_pick(buffer, chxact.ptime > 10, 3);

// Take 10 xacts while length of "buffer" is more than 5
chain_pick(buffer, buffer.length > 5, 10, to_kill);
```

chain_find - take xacts from user chain by index

- Prototype:

```
chain_find(
           word chainname,
           int xact_index,
           int count,
           word where_to_move (default is next block after that)
          );
```

- Usage:

  Since it is somewhat silly to remove xacts by their index (it is not such parameter which is always important or known), this block is generally used with *find()/find_minmax()* builtin functions (which return xact index when searching a xact in the user chain). Other behaviour is similar to *chain_pick* block - *count* or less xacts will be removed and every chain xact will be observed through check.

  Sometimes this block can be replaced with *chain_pick* block (and vise versa).

- Example:

```
// Finds 5 xacts which priority is less than 10 and removes them.
chain_find(buf, find(buf.xacts.pr < 10), 5);
```

hist_sample - add a sample to the histogram

- Prototype:

```
hist_sample(
            word histogram_name,
                int weight (default == 1)
           );
```

- Usage:

  This block, when executed, adds a sample (i.e. a value of parameter which is tracked by *histogram_name* histogram) to the specified histogram. Weight is a simple multiplier (how much samples should this particular sample be counted as).

- Example:

```
hist_sample(buffer_hist);
```

graph_sample - add a sample (X-Y pair) to the 2D graph

- Prototype:

```
graph_sample(
             word graph_name
            );
```

- Usage:

  This block, when executed, adds a sample to the specified 2D graph (i.e. it evaluates graph X and Y parameters and adds a pair of these values to the graph's values' table).

- Example:

```
graph_sample(time_queuelength_dependency);
```

- Additional hacks:

  - If sampled X is already present, new (X, Y) pair will be saved as (X, (oldY + Y) / 2).

while - do I really need to describe what it does..? :D

- Prototype:

```
while(
      expression condition
     )
{
        blocks which will be executed while condition is true
}
```

- Usage:

  This block will make xact loop through some blocks in curly braces while condition in parens is true. When it turns to false, xact will move further. Be aware of infinite loops!

- Example:

```
while(count < 10)
{
        wait(4);
        count += 1;
}
```

- Additional hacks:

  - This block can also be nested.

  - The process of iterating can be controlled by two blocks: *iter_next* and *iter_stop* (without empty parens after name). *Iter_next* forces xact to go to the next iteration and *iter_stop* forces xact to stop iterating at all (like *continue* and *break* in C).

loop_times - do something as much times as you need

- Prototype:

```
loop_times(
           word iterator,
           int upper_border
          )
{
```

```
        blocks which will be executed while iterator value is less than upper_border
}
```

- Usage:

  This block will make xact loop through some blocks in curly braces while *iterator* (it is the name of some variable or xact parameter) value is less than *upper_border* value. Iterator will be incremented automatically before every loop. **Un**like in other languages, you can change both iterator and upper border values while cycling, but, **like** in other languages, it can lead to awkward situations when done wrong.

  Inside loop *iterator* will consequently take values from its initial value to *upper_border* minus 1. After exiting loop, iterator value will be equal to *upper_border*.

- Example:

```
loop_times(xact.p1, 10)
{
        output("Xact p1 value is "+to_str(xact.p1));
}
```

- Additional hacks:

  - This block can also be nested.

  - The process of iterating can be controlled by two blocks: *iter_next* and *iter_stop* (without empty parens after name). *Iter_next* forces xact to go to the next iteration and *iter_stop* forces xact to stop iterating at all (like *continue* and *break* in C).

copy - make a full copy of a xact

- Prototype:

```
copy(
     int copies_count,
     word where_to_go (default == next to this block)
    );
```

- Usage:

  This block will make *copies_count* copies of a xact which is executing this block. If *where_to_go* mark is present, copies will be sent there. Every parameter will be copied except unique xact index.

- Example:

```
copy(4, tobuf);
```

- Additional hacks:

  - This block is useful in certain situations as it immediately adds xacts to the system with known group and parameters. Use it as an advantage.

  - Due to interpreter way of processing xacts, new xacts-copies will start to move only at the next beat.

output - print something when you need to

- Prototype:

```
output(
       int/float/bool/string/expression output
      );
```

- Usage:

If you need to print some values, info, debug messages while xacts are moveing throuth model, use this block. When xact will execute it, output message will be printed. Xact will simply move one line down. Output will be in such format: *(current time, current line, current xact index): your string*.

- Example:

```
output("Xact p1 value is "+to_str(xact.p1));
```

- Additional hacks:

  - Output string can be a whole expression with string (!) result or a simple number/boolean value.

xact_report - print all information about xact executing this block

- Prototype:

```
xact_report();
```

- Usage:

  This block will print all information about xact which is executing this block right now: group, index, line of code, all parameters including priority. You can use this for statistics gathering or debugging.

move - just skip that line

- Prototype:

```
move();
```

- Usage:

  When xact meets this block, it just moves to next line. That's all.

  This block is generally used inside other blocks in the interpreter, but you can also write it as some "foo" block in line which cannot be omitted but shouldn't do anything.

interrupt - force interpreter to go to next time beat

- Prototype:

```
interrupt();
```

- Usage:

  When executed, this block will send signal to interpreter to stop looking through CEC and just move to next time beat (which leads to taking some xacts from FEC if it's their time to leave and looking CEC from beginning).

review_cec - force interpreter to look through CEC from beginning

- Prototype:

```
review_cec();
```

- Usage:

  Just like *interrupt*, this block sends a signal to interpreter. CEC will be reviewed in the same time beat (interpreter won't move to next time beat).

- Additional hacks:

  - Following blocks automatically call *review_cec* inside them: *fac_leave*, *fac_goaway*, *fac_avail* and *reject* (and also every assignment to *xact.priority* parameter).

flush_cec - clear CEC entirely

- Prototype:

```
flush_cec();
```

- Usage:

  While *reject* presents a common and right way to delete xacts from model, *flush_cec* simply and rudely annihilates all xacts from CEC. Calling this block might be very dangerous for model's health. Facilities and queues are checked if they contain executing xact (but nothing is done for xacts which are deleted! Also, xacts in user chains and interruption chains are not bothered). Of course, after executing this block, current xact will stop its life too, and entirely system will halt until something arrives to CEC from FEC.

pause_by_user - halt simulation until user presses any key

- Prototype:

```
pause_by_user(
              string message (default == none)
             );
```

- Usage:

  When executing this block, interpreter will wait until user presses any key to continue. If *message* is present, it will be printed. You can use this block among with *output* and *xact_report* blocks while debugging.

- Additional hacks:

  - When using logging to file, this block might be strange to use because you will not know when to press any key - all output goes to file, you won't see these messages.

Built-in functions

Random generators:

- random_int(min, max) - generates a pseudo-random integer between *min* and *max*, including min and max values.
- random_float(min, max) - generates a pseudo-random floating point value *min* <= value <= *max*.
- random01() - generates a pseudo-random float value between 0 and 1 (like probability).

Type converters:

- to_str(val) - tries to convert *val* into string and returns string.
- to_int(val) - tries to convert *val* into integer and returns integer. Can raise "cannot convert" error.
- to_float(val) - tries to convert *val* into floating point number and returns it. Can raise "cannot convert" error.
- to_bool(val) - tries to convert *val* into boolean value (true/false). Following values can be converted: true/false boolean values, "true"/"false" strings, any numbers (==0 - false, !=0 - true); otherwise, error will be raised.

find() - find name of struct by condition connected to struct's parameter

- Prototype:

```
find(
     condition (looks like "structure.parameter ==/>/<... expression")
    );
```

- Usage:

  In situations where you need to know name of free facility, chain with known length, etc., *find()* can help you.

First word at the left of conditional expression is "facilities"/"queues"/"chains"/"chains.xacts"; second word after dot can be any available parameter name for according structure. According to first word, return value will differ: if you search for a facility/queue/chain, it will be first suitable structure's name (and "", if nothing was found); if you search for xact in chain, it will be first suitable xact's index (and -1, if nothing was found).

- Examples:

```
find(facilities.curplaces > 0) ==> returns name of facility
find(chains.length < 10) ==> returns name of chain
find(chains.xacts.p2 == 5) ==> returns xact index
```

find_minmax() - similar to find(), but finds struct with min/max value of its parameter

- Prototype:

```
find_minmax(
            word min/max,
            name of the structure and its parameter
           );
```

- Usage:

  This function acts as *find()* function, but it doesn't actually operates with a condition. Instead, it will execute search of a structure (in the range of given names, for example, *facilities.curplaces*) to end up with a structure with minimal or maximal value of its parameter. Again, return value depends on what search do you perform.

- Examples:

```
find_minmax(max, facilities.curplaces) ==> returns name of the facility
find_minmax(min, chains.length) ==> returns name of the chain
find_minmax(max, chains.xacts.pr) ==> returns index of xact
```

Math functions:

- abs_value(number) - returns an absolute value of int/float number.
- exp_distr(x, lambda) - returns value of a cumulative exponential distribution function with given x and lambda (more on Wikipedia). Is useful when setting processing times, etc.
- round_to(value, digits) - returns *value* rounded to floating number with *digits* digits after point (*digits* default value = 0).

Simulation results

When simulation is finished, a lot of information about model is printed on the screen. It includes listing of the program which was simulated, simulation time, values of all variables, values of parameters and statistics from every structure of the model (facilities, queues, chains, etc.) and contents of CEC and FEC.

Statistics is a very important aspect in languages like GPSS and OpenGPSS (that's why we're simulating these models at all - we want to know some information about how they are behaving), so, here's the list of what you can find in simulation results:

- Program listing

  If you will take a look at listing, you'll see three numbers at the left of each line (where there's no number it means that this number equals zero). First is line index, second - how much xacts are currently on this line (for example, blocked before facility), third - how much times this line was executed (how many xacts passed this line).

- Variable values

- Listing of facilities

  Every facility gathers lots of statistics:

  - max xacts - maximum amount of xacts which were in facility simultaneously (*places* shows capacity of facility, so, max xacts can be less or equal than places).

  - auto queued - if this facility has an automatically created queue. All queues are shown below facilities.

- enters - how many xacts entered this facility.

- busyness (weighted and unweighted) - it is a proportion which shows how much time this facility was working (i.e. was occupied at least by one xact). Weighted busyness goes from 0 to 1, it doesn't depend on facility's places. Unweighted busyness goes from 0 to *places*.

(How it works: every beat current amount of xacts in facility is summed, and in the end this sum is divided by number of beats.)

- current xacts - shows a list of currently processing xacts.

- average (xact) processing time

- is available - shows status of facility (which can be changed by fac_avail and fac_unavail blocks).

- avail time and unavail time - number of beats in which facility was available/unavailable.

- availability - just a proportion of previous values.

- irrupted xacts - it is a length of interruption chain in this facility (i.e. how many xacts wait to return back to processing).

- irruption chain

- Listing of queues

  Every queue also gathers lots of statistics:

  - enters - how many xacts have been queued

  - zero entries - how many times xact enters this queue when it was empty

  - max, average and current length (of the queue)

  - average xact waiting time (in the queue) - average time from entering to leaving this queue for each xact.

  - average time without zero entries - since zero entry means waiting time was 0, we can recalculate this parameter without zero entries (if there is "no data", it means every time xact enters this queue it was empty).

  - max xact waiting time (in the queue)

  - list of currently queued xacts

- Listing of chains

  Current length and contents of chain are printed.

- Listing of marks

  Mostly for debug purposes, every mark is printed along with its corresponding line.

- Histograms and graphs

  These are a special way to gather statistics of the model. All sampled values for histogram or graph will be printed. For histograms some statistics also will be printed, such as entries for each interval, percentage of these entries and pseudographical representation of histogram.

- Future Events Chain and Current Events Chain

  Again, for debug purposes, you can look inside CEC and FEC of the model at the moment when simulation has stopped.

Errors and warnings

When something went wrong, OpenGPSS interpreter will interrupt simulation and print an error message to console. Also some warnings can be printed (warnings do not interrupt simulation) if there's something not very right in your model and you probably want to know about it.

Along with error index and message, line index (if any can be specified) is printed. Remember, it is index of line in interpreted program, not in .ogps file (so when you got index of line with error, you need to find listing of a program in the beginning of current log).

List of errors:

- **1**: No such file: "name of file"

  Name of file to interpretet is specified incorrectly. It must be without .ogps extension and, if file is not in the same folder where interpreter is, absolute or relative path should be specified.

- **2**: Unexpected end of file during parsing program into lines

  Probably, incorrect last line (";" missing, etc.)

- **3**: name of the definition expected; got "type" "contents"

  After definition keyword (like int, fac, hist<> and so on) there must be name of the definition.

- **4**: Unknown parameter "parameter" or missing "}" during initialization

  There is either unknown word or incorrect construction (something not like "known_word = initial_value") in curly braces after definition's name.

- **5**: Expected initial value of type "type" for parameter "parameter"; got "token"

  Type mismatch, read manual page for this definition.

- **6**: Expected name of defined variable/structure/array or value; got "token"

  Unknown word in the program (mostly possible - a typo).

- **7**: Cannot "inc/dec"rement string or boolean

  Check types, again, for assignments.

- **8**: Cannot apply operation "operation" for string values

  Same.

- **9**: Found incorrect float number "number" during analysis

  Probably misspelling, dot instead of comma, etc. Correct float number is "number.number".

- **10**: Initial value for integer variable "name" must be of type "int"

  Again, type error.

- **11**: Initial value for float variable "name" must be a number

  Same.

- **12**: Nothing or "something" expected; got "token"

  Unexpected constructions in the program (misspelled words, inexisting operators, etc. Just check line with error).

- **13**: Mark "name" found more than once as transporting label

  Marks address lines in your program; every mark can represent only one address (like in real life, streets in one city cannot have matching names).

- **14**: Xact is trying to leave executive area

  Interpreter founded out that next block is "}}", or it is not in executive area at all. Probably, no *reject* block in the right place or transport argument was incorrect.

- **15**: Unknown word "word" used as mark name

  There's something at the left of ":", and it is not a mark name.

- **16**X: "," or ")" expected while parsing arguments; got "token"

  You wrote something wrong in parens of a function. "A" - function is attachable, "B" - built-in, "C" - conditional.

- **17**: String "xact group name" expected; got "token"

  Again, type error (most of the time) or misspelling.

- **18**: Number argument expected; got "token"

  Same.

- **19**: Expected "type" value for parameter "name"; got "token"

  Same.

- **20**: Cannot do addition between string and non-string value

  And it is right. If you want, use built-in type conversion functions.

- **21**X: "token" expected; got "another_token"

  Some error during parsing which parser could not handle. May be an absence of some required token or something like that.

- **22**: Multiple definition of name "name" with type "type"

  Different variables/structures of the same type have matching names.

- **23**: Exit condition must be declared only once

  And it really must be.

- **24**: Index of executive line is out of bounds (probably missing "}}")

  You forget to close executive area, and interpreter went out of program's bounds.

- **25**: Current xact from group "group" does not have a parameter "name"

  Well, you made something wrong while initializing parameters and guiding xacts in the model.

- **26**: Unknown structure value "name" (are you trying to assign to read-only values?)

  All values of structures are read-only, since model and statistics depend on them. This error happens every time you try to assign to something structural which is not a xact parameter.

- **27**: Unknown variable "name"

  You specify assignment to a "name", but interpreter cannot find this variable.

- **28**: Cannot take name of "var/struct", because it is unknown

  Again, specified name is unknown, interpreter cannot do anything with it.

- **29**: Error while transporting; undefined mark "name"

  Some blocks have a transporting feature, and mark name will be checked before transporting, generating this error.

- **30**: Mark "name" is not present anywhere as transporting label

  This mark is defined, but it doesn't point to any line of a program. (You will also see warning about this mark; but warning is printed before simulation starts, and error is printed when you actually try to use this mark as label.)

- **31**: Cannot convert "var" to "type"

  You are trying to make impossible with the help of type conversion functions.

- **32**: Initial value for boolean variable "name" must be true/false word

  It's all said in the error message.

- **33**: Cannot perform "operation" for types "" and ""

  Again, type mismatch.

- **34**: What type of transport is here? Expected ">", "|" or "?", got "token"

  "->" is written, but there's nothing after it.

- **35**: Condition or probability argument is missing

"->|" or "->?" is written, but only one argument is specified.

- **36** (and **37**): "}" for if/else_if/else/while/loop_times is missing

  Interpreter cannot find closing brace for one of these blocks.

- **38**: Cannot find the owner of a "token" in line ""

  Interpreter tried to find whose brace is there, but there's no branching/looping block right above it.

- **39**, **40**, **41**, **42** (and also **47** and **49**): Xact is trying to do something with paired blocks (***_enter/leave/irrupt), and he double entered/double leaved facility/queue or interrupted facility he's already in, and so on.

  So, be careful when using paired blocks - xact (most of the time) must **not** jump from/to blocks this pair bounds! (or do this, but be very attentive)

- **43**, **44**, **48**, **53**, **59**: no such structure (facility/queue/chain/histogram/graph): "name"

  Block was given with incorrect structure name (check what can be a result of your expression, if you use them).

- **45**: Expected ","; got end of line during parsing "loop_times" block

  loop_times takes two arguments - iterator and bound value.

- **46**: Cannot do assignment; "var" is a read-only value

  There are some variables in the system which are read-only, do not try to assign to them.

- **50**: Unknown parameter "name", cannot execute search for it

  Check what parameter are you trying to search by in find/find_minmax functions.

- **51**: Expected parameters for histogram initialization

  You hadn't specified histogram parameters in curly braces (they cannot be omitted).

- **52**: Some parameters for histogram initialization are missing

  You hadn't specified all needed parameters.

- **54**: Wrong function definition (expecting one condition per each return expression)

  Read manual page about defining conditional functions or double-check your definition.

- **55**: Wrong number of arguments for function "name"

  As it says...

- **56**: Array index is out of range (same error for matrices)

  Maybe you iterate too much further, or expression is given you wrong index as a result.

- **57**: Cannot find attachable module "name"

  All attachable modules must be in root folder of an interpreter. (or, maybe you just misspell?)

- **58**: Attachable module does not have a function "name"

  Again, may be misspelling.

- **60**: Unknown configuration parameter "name"

  Do not ever write something in config file except True/False words after "=". Ever.

- **61**: Multiple variable/array definition with the same name "name"

  Variables, even if they're of different types, cannot have matching names (see "About name ambiguity" section of this manual).

- **62**: Arrays/matrices of type "type" are not allowed

  You cannot create arrays of functions and marks (because you don't need to).

- **63**: Value of parameter "name" is of unknown type

  You set a wrong value for the specified configuration parameter.

List of warnings:

Their messages speak for them, so there'll be no explanations.

- **1**: Everything except definitions in non-executive area will be totally IGNORED. If you see this, double-check definition area of your program.
- **3**: Mark "{}" cannot be found as transporting label (at the left of ":"). Is it needed at all?'
- **4**: Duplicate xact parameter "{}"; previous value of "{}" will be OVERWRITTEN.
- **5**: Attachable module "{}" is either already imported or its name conflicts with one of other attachable modules. Double-check "attach" statements in your program.